Detecting Contradictions from CoAP RFC Based on Knowledge Graph

Xinguo Feng[1(✉)], Yanjun Zhang[2], Mark Huasong Meng[3,4], and Sin G. Teo[3]

[1] The University of Queensland, St. Lucia, Australia
s.feng@uq.edu.au
[2] Cyber Security Research and Innovation (CSRI), Deakin University, Geelong, Australia
[3] Institute for Infocomm Research, A*STAR, Singapore, Singapore
[4] National University of Singapore, Singapore, Singapore

Abstract. Due to the boom of Internet of Things (IoT) in recent years, various IoT devices are connected to the internet and communicate with each other through web protocols such as the Constrained Application Protocol (CoAP). These web protocols are typically defined and described in the Request for Comments (RFC) documents, which are written in natural or semi-formal languages. Since developers largely follow the RFCs when implementing web protocols, the RFCs have become the *de facto* protocol specifications. Therefore, it is desirable to ensure that the technical details being described in the RFC are consistent, to avoid technological issues, incompatibility, security risks or even legal concerns. In this work, we propose RFCKG, a knowledge graph based contradictions detection tool for CoAP RFC. Our approach can automatically parse the RFC documents and construct knowledge graphs from them through entity extraction, relation extraction, and rule extraction. It then conducts an intra-entity and inter-entity consistency checking over the generated knowledge graph. We implement RFCKG and apply it to the main RFC (RFC7252) of CoAP, one of the most extensively used messaging protocols in IoT. Our evaluation shows that RFCKG manages to detect both direct contradiction and conditional contradictions from the RFC.

Keywords: Contradiction detection · Knowledge graph · Natural language processing · Request for comments · IoT protocols

1 Introduction

The Internet of Things (IoT) is an emerging technology in recent years. It refers to “devices and sensors” that are uniquely addressable based on their communication protocols, and are adaptable and autonomous with inherent security [5]. Its development is closely connected to many cutting-edge technologies such as blockchain [10,13], smart home [17], and machine learning [11,23,32]. During the past decade, IoT applications have experienced rapid growth and have been

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
X. Yuan et al. (Eds.): NSS 2022, LNCS 13787, pp. 170–189, 2022.
https://doi.org/10.1007/978-3-031-23020-2_10

successfully applied on both individual (e.g., e-health, smart home) and professional levels (e.g., smart supply chain, smart city, industry IoT) [16,28]. It is estimated that there will be over 500 billion IoT devices connected to the Internet by 2030 [25].

Similar to traditional web endpoints, IoT devices communicate through the corresponding *web protocols*, which are defined by the Request for Comments (RFC) specification documents. An RFC is a specification document that describes the technical details of a web protocol. However, it is challenging to perform formal verification on an RFC because it is usually written in natural human languages. There might exist some *contradictions* and *inconsistencies* in an RFC that cannot be easily spotted. Furthermore, ambiguities might be introduced in the writing of an RFC when people have different understandings or interpretations of the protocol design. All of those issues may lead to confusion for users who want to utilise the protocol in their implementations. Take RFC7252 [24] on The Constrained Application Protocol (CoAP) as an example. Californium [12] is a significant Java implementation for CoAP. On its GitHub repository, 170 out of 2,030 issues in total (around 8%) mention the keyword RFC[1], indicating there might be discrepancies between the implementation and the description in the RFC. Those discrepancies are possibly caused by the contradictions existing in the RFC itself. Considering such contradictions may make communicating devices malfunction and introduce potential security issues, we see the need to validate if the defined technical details are consistent within the specification document.

Existing work on RFC or similar specification documents focuses on extracting finite state machines to perform security analysis dynamically [22,30]. There is limited work on detecting contradictions in natural language documents themselves, especially in specification documents like RFC. We seek a way to fill this gap. In this paper, we propose RFCKG, an approach to construct a knowledge graph from RFC documents and detect potential contradictions from the knowledge graph. We construct the knowledge graph through entity extraction, relation extraction, and rule extraction with NLP techniques, such as coreference resolution, sentence split and dependency parsing. We apply RFCKG to RFC7252 of CoAP. It manages to detect one direct contradiction and four conditional contradictions from the RFC.

We summarise our contributions as the following:

- We identify the essential components for constructing a knowledge graph from RFC documents, such as entity, relation, and rule.
- We define two types of contradictions and propose a general framework to check for them with the constructed knowledge graph.
- We propose RFCKG, a general framework to construct a knowledge graph with the extracted components of an RFC document. We demonstrate that RFCKG can successfully capture contradictions from a real-world RFC, which shows the soundness of our approach.

[1] Assessed on 11th August, 2022.

2 Background and Related Work

In this section, we introduce some background knowledge and related work that inspire this work.

2.1 Background

RFC. An Request for Comments (RFC) is a specification document that describes the technical details of a web protocol. It is usually written by engineers or computer scientists to describe the methods, behaviours, or innovations of the web protocol in natural human languages. Developers who wish to implement the protocol or users who wish to utilise the implementations should always refer to the RFC that defines the protocol.

Knowledge Graph. Since the technical details are all described in the specification documents, extracting the knowledge and representing it in an appropriate data structure is desired. A knowledge graph (KG) is a multi-relational graph composed of *nodes* (entities) and *edges* (relations), and each edge can be represented in a triplet (head entity, edge, tail entity).

2.2 Related Work

Knowledge Graph Representation for Documents. Li *et al.* [15] construct a knowledge graph from API documents, which can be easily accessed by developers. A knowledge graph is a multi-relational graph constructed with nodes (entities) and edges (relations), where each edge indicates the two entities are connected by a specific relation [31]. Mondal *et al.* [21] propose a way to do an end-to-end knowledge graph construction on NLP related papers to describe NLP tasks and their evaluations. Typical tasks for constructing a knowledge graph are entity extraction and relation extraction. Although there exist some tools [9,26,35] for these tasks, they are usually for general purposes. It is unlikely that they would work well with tasks that are domain-specific without further injecting the domain knowledge.

Rule Extraction for RFC. Rules in RFC define the functionalities and behaviours of the protocol. The natural language writing style of rules is specified in RFC 2119 [3]. In particular, it defines the modal keywords to indicate the requirement levels for rules [27]. Furthermore, RFC 8174 [14] emphasises the usage of uppercase letters for modal keywords defined in RFC 2119. Tian *et al.* [27] extract the rules with keyword matching and use dependency parsing to process the rules. Dependency parsing is also present in other work such as [33]. It works well with simple sentences but suffers with complicated sentences with multiple objects or multiple subordinate clauses.

Different Representations for Specifications. Andow *et al.* [1] construct an ontology on applications' privacy policy documents and check for logical contradictions (e.g. "Not collecting personal information" contradicts with "but collecting email address"), which is the main inspiration for this work. Wang *et al.* [30] utilise traffic, documents, and configurations of several IoT protocols and construct the finite state machines to evaluate their security. Pacheco *et al.* [22] also construct finite state machines of protocols from documents to perform attack synthesis.

Contradiction Checking for Specifications There is limited work on discovering contradictions in the text. Harabagiu *et al.* [8] propose an approach to recognise negation, contrast and contradictions for general text. Xie *et al.* [34] study the privacy policy compliance issue of virtual personal assistant apps. Wang *et al.* [29] propose a formal analysis framework to detect specification contradictions in single sign-on protocols. Mahadewa *et al.* [18] explore contradictions in privacy policy on trigger-action integration platforms. Another recent work by Meng *et al.* [19] proposes a systematic analysis methodology to scrutinize the compliance of mobile operating systems in protecting users' privacy. However, recognising contradictions in specifications documents, such as RFC, has not been well studied.

3 Problem Definition

RFCs are written in unstructured natural languages. RFCKG parses them and generates knowledge graphs that can be automatically checked for contradictions. In this section, we first define the components of the knowledge graph (Sect. 3.1), then we present the types of contradictions we target to detect in this work (Sect. 3.2).

3.1 Entity, Rule and Relation

The knowledge graph generated by RFCKG consists of three components, i.e., *entity*, *rule* and *relation*, where the *entities* and the *rules* are represented as nodes, and the *relations* are represented as edges.

Entity. We refer an entity in RFCs as an object in web protocols that has functionalities or behaviours being described. Commonly used entities include "message", "options", "token", etc. We use a single field data structure to represent an entity node in the knowledge graph: (*name*). For example, the entity "confirmable message" is represented as (*confirmable message*).

Rule. A rule node consists of a set of atomic rules appearing in a same rule statement, concatenated by logical connective "$\wedge$ (AND)", "$\vee$ (OR)", "$\veebar$ (XOR)" and "$\neg$ (NEGATION)". We define an atomic rule as a four-tuple data structure:

$$(\{variable, operator, value\}, necessity),$$

in which $\{variable, operator, value\}$ represents the rule content, and the *necessity* represents the requirement level, including "STRONG" and "WEAK", where "STRONG" indicates an absolute requirement level such as "MUST", "REQUIRED", "SHALL", "MUST NOT", and "WEAK" indicates an optional requirement such as "NOT RECOMMENDED", "MAY" and "OPTIONAL". For example, in the statement: *"Message version number MUST be set to 1 and the options of the message MUST be cached"*, the extracted rule is:

$$(\{version_number = 1\}, STRONG) \wedge (\{cached_options = TRUE\}, STRONG)$$

Relation. The relations RFCKG targets to extract from RFCs include (1) the relation between an entity and an entity, e.g, "A version number [entity] is a field of [relation] a message [entity]", (2) the relation between an entity and a rule, e.g., $(\{version_number = 1\}, STRONG)$ [rule] is a rule of [relation] confirmable message [entity], and (3) the conditional relation between a rule and a rule. For example, in the statement *"If the version number of a message is not set to 1, the options of the message MUST NOT be cached"*, $(\{version_number != 1\}, STRONG)$ [rule] is a condition of [relation] $(\{cached_options = FALSE\}, STRONG)$ [rule]. The former is the antecedent rule and the latter is the consequent rule.

Figure 1 illustrates the KG representation of the rule statement "If the version number of a message is set to 1, the options of the message MUST NOT be cached".

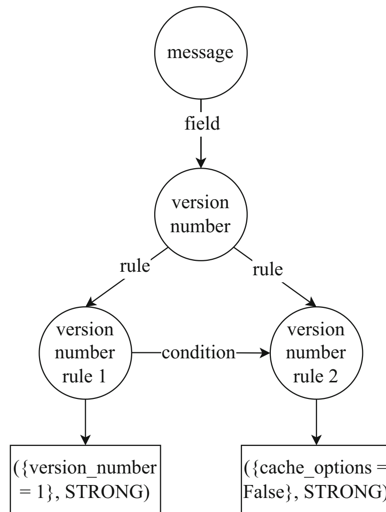

Fig. 1. RFCKG's knowledge graph representation of an example statement in RFC: "If the version number of a message is set to 1, the options of the message MUST NOT be cached."

3.2 Contradictions

The core idea of RFCKG is to represent an unstructured RFC document under analysis as a structured knowledge graph, and then to check its rules for contradiction detection. In particular, we define two types of contradictions as follows:

- **Direct contradiction.** This occurs when different rules of a same entity e - denoted as $\{r_1, ..., r_n\}_e$ contradicts with each other. That is, the conjunctions of rules is evaluated as false, i.e.,

$$\bigwedge_{i=1}^{n} \{r_i\}_e = FALSE$$

  A direct contradiction is regarded as an erroneous inconsistency of an RFC which may lead to implementation issues. This contradiction captures the following three scenarios.

  (1) Contradiction among plain rules. A plain rule refers to a rule that is not an antecedent rule or consequent rule. For example, consider the following rule statements *"The version number of a message MUST be set to 1"* and *"Message version number MUST be 0"*. The rules for these rule statements are ($\{version_number = 1\}$, STRONG) and ($\{version_number = 0\}$, STRONG). These are plain rules as they are not antecedent rules or consequent rules. We concatenate them and see that they evaluate as false.

$$(version_number = 1) \wedge (version_number = 0) = FALSE$$

  (2) Contradictions between plain rule and consequent rule. It occurs when a plain rule states "A must be True", while a consequent of a conditional rule states "A is False". For example, consider the following rule statements *"The version number of a message MUST be set to 1"* and *"If the options of the message are cached, the version number of the message MUST be set to 0"*. The rule extraction and evaluation are the same as in the example above.

  (3) Contradictions between the consequent of conditional rules. For example, two conditional rules state the same antecedent while implying a contradicted consequent. For example, consider the following rule statements *"If the options of the message are not cached, the version number of the message MUST be set to 1"* and *"If the options of the message are not cached, the version number of the message MUST be set to 0"*. The rule extraction and evaluation are the same as in the example above.
- **Conditional contradiction.** This occurs when the antecedent of conditional rules contradicts others. Denote c_i as an antecedent rule of a conditional proposition, RFCKG reports a conditional contradiction if

$$\bigwedge_{i=1}^{n} \{r_i, c_i\}_e = FALSE$$

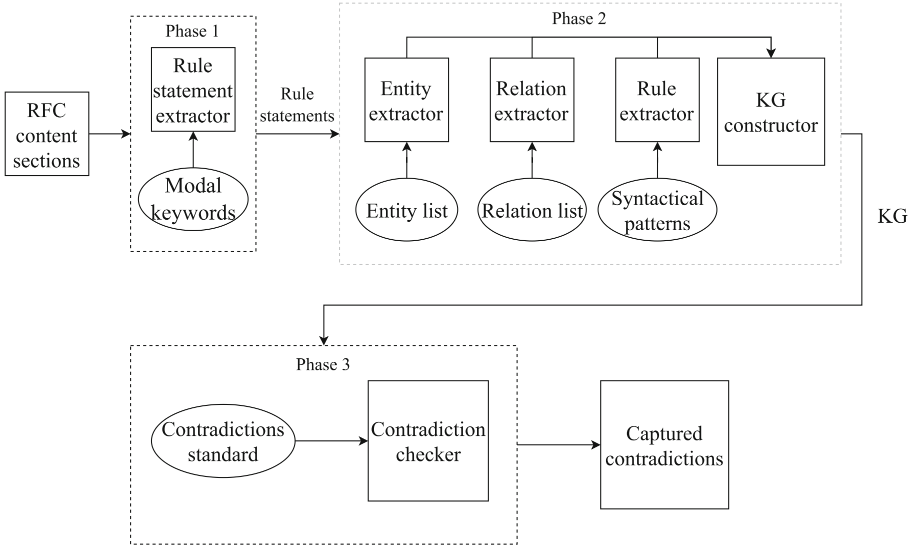

Fig. 2. Overview of approach showing three phases

For example, consider the following rule statements *"Message version number SHOULD be set to 1"* and *"If the version number of a message is not 1, the options MUST be cached"*. The rule for the first statement is ({*version_number* = 1}, STRONG), which is a plain rule. The antecedent rule for the second statement is ({*version_number* != 1}, STRONG). We concatenate them and see they evaluate as false.

$$(version_number = 1) \wedge (version_number\, ! = 1) = FALSE$$

The conditional contradictions extracted by RFCKG can highlight the instruction of error handling in RFC, which is likely ignored by the developers especially when such statements appear in different places in the document.

4 RFCKG Approach

We design RFCKG as a three-phase approach that consists of *rule statements extraction*, *knowledge graph construction* and *Contradiction detection*, as shown in Fig. 2.

4.1 Rule Statement Extraction

This phase aims to extract statements that contain rules of interest from RFCs. We first remove the irrelevant information from the document including the headers, list of content, acknowledgement, references, figures, and tables. We then use the `Natural Language Toolkit (NLTK)` [2] to split the rest of the document into sentences.

Table 1. Modal keywords for extracting rule statements

STRONG keywords	WEAK keywords
MUST, REQUIRED, SHALL, MUST NOT, SHALL NOT	SHOULD, RECOMMENDED, SHOULD NOT, NOT RECOMMENDED, MAY, OPTIONAL

Since the release of RFC2119 (which specifies the standard for keywords usage in RFCs to indicate requirement levels) in 1997 [3], RFC documents (released after 1997) enforce the use of capitalized modal verbs (such as "MUST", "MAY" and etc.) to indicate the requirement level of a rule in the specification. We therefore examine the capitalized modal verbs used in the sentences, and identify a sentence that contains those capitalized modal verbs as a rule statement. More specifically, we extract the strong statements and weak statements based on the modal keywords as shown in Table 1 following the definition of prior work [27]. Algorithm 1 in Appendix B demonstrates the details for rule statements extraction.

4.2 Knowledge Graph Construction

This phase aims to identify *entities*, *rules* and *relations* based on the extracted statements, and represent them in a knowledge graph.

4.2.1 Entity Extraction

To identify the entities, we select 56 types of common entities from summarising the terminologies from the Terminology section of RFC7252 as shown in Table 5 in Appendix A. Given a rule statement, if an entity in the predefined list appears in the statement, we add it to our extracted entity list.

4.2.2 Rule Extraction

There are seven steps for extracting rules from rule statements, which are rule context construction, rule entity determination, co-reference resolution, sentence splitting and rephrasing, conditions identifications, rule construction, and variable normalisation.

- **Context construction.** For each rule statement we extract, it is one complete sentence. To better address the later tasks, we need to construct a semantic context environment for each rule statement. For each rule statement we extract, we locate its position in the RFC document and backtrack five sentences before it. This is based on the assumption that, if a rule statement describes an entity's behaviours, the neighbouring sentences before it should also mention the same entity.
- **Rule entity determination.** When we construct the Rule nodes in a later step, we need to connect each Rule node to a corresponding Entity node. For each rule statement context, we iterate through the extracted entity list and count the occurrences of each entity. We take the entity that appears the most

as the rule entity. If there are multiple entities that have the same number of occurrences, we take the one that appears the last as the rule entity.

- **Co-reference resolution.** It is common in natural languages to use co-references to refer to words or phrases that are mentioned before. We aim to find the co-references in the rule statements and substitute them with the actual words or phrases they are referring to, so that we have complete and rich information in each rule statement for the next step. We use the co-reference resolution functionality in the `spaCy` [9] NLP tool to address the pronoun co-reference, such as "it", "them", etc. For other co-references that the tool cannot address, such as "this field", we use the rule entity we identify for each rule to substitute them.
- **Sentence splitting and rephrasing.** A rule statement is a sentence that describes one or several behaviours of an entity, which means the structure of the sentence can be complex. It would be easier to process the rule statement if we can split one complex sentence into multiple simple sentences but retain the semantics that describes the behaviours, so that we can process these simple sentences one at a time. To address this, we use the dependency parsing functionality in the `spaCy` [9] NLP tool and look for the root of the rule statement, then look for words that have a conjunction dependency relation with it. We then look for the subject of the rule statement, split the rule statement with the conjunction words, and concatenate each of the split sentences with the subject in the front. In this way, we split and rephrase the complex rule statement into multiple complete but simple rule statements. To determine the logical connective between the split sentences, we see if the keyword "and" or "or" appears in the original rule statement. If "or" appears, we determine the logical connective to be "$\vee$ (OR)". If not, we determine the logical connective to be "$\wedge$ (AND)".
- **Condition identification.** Recall that we define a type of contradiction as conditional contradiction in Sect. 3.2. For a rule statement, we need to know if there exists a conditional relation between different behaviours. To address this, we look for rule statements that start with the word "If", and split it at the first comma. We give the first part an *antecedent* label to indicate it is an antecedent rule, and the following part a *consequent* label to indicate it is a consequent rule. For the other situation that we describe above, we give the split sentences an *entity* label, indicating that they are plain entity rules.
- **Rule construction.** Recall that we define an atomic rule as a four-tuple data structure:

$$(\{variable, operator, value\}, necessity),$$

  We construct atomic rules on a split sentence level. The rules we try to construct are operations that the entity performs to describe its behaviours. There are two main types of atomic rules. The first type specifically describes

that an item is set to, equal to, larger, or smaller than a value. The other type describes an operation being performed, but does not specifically describe any value. We review part of the split sentences and define 59 syntactical patterns for `spaCy` [9] to extract variables for constructing the atomic rules. A syntactical pattern is a pattern that describes the dependency relations between the components we want to extract.

For example, consider the following rule statement *"Implementations of this specification MUST set the version number to 1"*. The variable we want to extract from the first example above is "*version_number*". We look at the dependency relations of this sentence, as shown in Fig. 3. We define the pattern as *{verb, dobj, compound, prep, pobj}*, indicating that we want to extract the verb, the direct object of the verb, the compound of the direct object, the preposition of the verb, and the prepositional object of the preposition. The information we extract with this pattern from this sentence is *{set, number, version, to, 1}*. We organise the order of these words and construct the atomic rule as *({version_number, =, 1}, STRONG)*, in which the necessity "STRONG" comes from the fact that the original rule statement is a strong rule statement.

For each split sentence that belongs to a rule statement, we apply these 59 defined patterns to them and construct atomic rules accordingly. Then we apply the logical connective that we extracted above to concatenate these atomic rules to construct a Rule object that represents the original rule statement. The *entity*, *antecedent* and *consequent* rule labels are carried over to these constructed rule objects.

- **Variable normalisation.** The 59 defined syntactical patterns are applied to all split and rephrased sentences, as we want the defined syntactical patterns to be able to generalise to more sentences that have similar structures. There could be situations where variables extracted from different patterns actually mean the same thing. Also, different words might have the same or similar meanings. These different variables that actually could mean the same should be grouped as one variable as it might affect the accuracy of the contradiction checking later. To address this, for each Entity node, we gather all the variables of all the Rule nodes under it. For each variable, we use the `spaCy` [9] tool with its internal word embedding to get the variable's average embedding vector. Then we compute the cosine similarity between all variable pairs. If the similarity is larger than 0.9, we mark them as similar variables. After we gather all the similar variables, we substitute the variable that has the longer name with the shortest one between them.

4.2.3 Relation Extraction

For the entity-entity relation extraction, we examine each combination of the entity pair we extracted and identify the relation defined in Table 5. For the entity-rule relation extraction, recall that we already identify that each rule should belong to an entity. We simply use the "rule" relation defined in Table 5 in Appendix A. For the rule-rule relation extraction, if a rule has the *antecedent* rule label and the

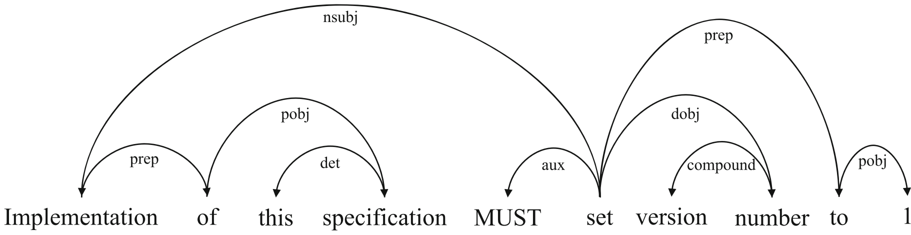

Fig. 3. Dependency relations of the example rule statement "Implementations of this specification MUST set version number to 1"

following rule from the same original rule statement has the *consequent* rule label, we use the "condition" relation defined in Table 5 in Appendix A.

4.2.4 Graph Representation

We use the open source library `NetworkX` [7] for the graph construction. We first create Entity nodes for the extracted entities and add them into the graph. We then create directed edge between Entity nodes and connect them with the corresponding relation as the edge label. For example, as a "confirmable message" [entity] is a type of [rule] a "message" [entity], the two entities are first added into the graph and an edge with the label "type" is then added which points from "message" to "confirmable message". We go through all the extracted entity pairs and their relations, and a skeleton graph (without Rule nodes) is constructed.

We then construct the Rule nodes and their corresponding entity-rule and the rule-rule edges. For each Rule node, we first add it to the graph. Then we create a directed edge with a "rule" label pointing from its possessor Entity node to itself. For example, Rule(rule={"version_number:=": 1}, necessity="STRONG") belongs to Entity(version_number). We complete the entity-rule edges construction at this state. Then we examine each Rule node and see whether it carries an *antecedent* rule label from the previous steps. If it does, we locate the corresponding Rule node that carries the *consequent* rule label and create a directed edge with a "condition" label pointing from the antecedent Rule node to the consequent Rule node.

4.3 Contradiction Detection

We now describe the contradictions detection step with the knowledge graph constructed described above. We do contradictions detection on the entity level. Recall that each atomic rule within a Rule node is constructed with the variable, the operator and the value. When checking for contradictions for an entity, we concatenate the atoms within the same Rule node with the corresponding logical operator as the Rule node's expression, then we concatenate the different Rule nodes' expressions under the same Entity node with the "$\wedge$ (AND)" operator. We then use the open source Python library `SymPy` [20] as a solver to see if it is consistent (return True of False).

To prepare for contradiction checking, we need to transform the variables and values into appropriate forms, so that the solver can construct the expressions and evaluate if they are consistent or not. For each Entity node in the graph, we traverse the graph to get all the Rule nodes that it is connected to, and separate them into entity rules, antecedent rules and consequent rules. We collect all the unique variables we extracted in the three rule sets and create a unique symbol for each of them. We also collect all the unique values for these variables. There are multiple value types for the values, such as numerical, string, and boolean. We keep the numerical values as they are and transform the other value types into unique integer values for the solver as the solver can only accept numerical values. We start by taking a seed integer, for example, 10,000, and we iterate through all the values that are not numerical. If we found a value that is not transformed yet, we assign this seed integer to it and increment the seed by 1 as the new seed, and repeat the process. To determine the appropriate value of the seed, we simply go through all the unique numerical values and pick one that does not collide with the existing values. For the operators “>”, “>=”, “=”, “!=”, “<=”, and “<”, we keep them and call the corresponding evaluation functions in the `SymPy` [20] library.

We check for direct contractions first. Recall that there are three types of direct contradictions. For the first type of contradiction, which is a contradiction between plain rules of an Entity node, we take each variable in each Rule node in the entity rules set and create an atom logical expression for it with its corresponding operator and value. We store this expression and iterate the next variable. We create another atom logical expression for the next variable and concatenate it with the previous one with the logical connective described in the Rule node. After iterating all the variables in the Rule node, we have a final expression that represents this particular Rule node. We store this final expression then iterate to the next Rule node and repeat the process. We then concatenate these two expressions that represent two different Rule nodes with “$\wedge$” as these are plain entity rules, and the variables in them should all be consistent. If it is evaluated as true, we store the concatenated expression and iterate to the next Rule node and repeat the process. If it is evaluated as false, then we find a direct contradiction. We record the contradiction and remove the second expression from the concatenated expression and iterate to the next Rule node for further contradiction checking. Algorithm 2 in Appendix B demonstrates this process.

The contradiction detection for the other two types of direct contradictions are similar. For the second type, we keep the final expression that is evaluated as true from when we check for the plain entity rules, and iterate through all the consequent Rule nodes in the same process described above. For the third type, we only iterate through the consequent Rule nodes that are pointed from the same antecedent Rule node without the final entity rules expression, in the same process described above. For any direct contradiction found, if both original rule statements have the same requirement level (both strong or weak), we cannot

Table 2. Results on knowledge graph construction from RFC7252

Rule statements			Graph nodes			Rule nodes			
Strong	Weak	Total	Entity	Rule	Total	Plain	Antecedent	Consequent	Total
136	81	217	28	319	347	220	41	58	319

recommend which one to follow. If not, we can recommend to follow the one with a strong requirement level.

We then check for conditional contradictions. The process is also similar to direct contradiction checking. We keep the final expression that is evaluated as true from when we check for the plain entity rules. We then iterate through each Rule node in the conditional rules set, construct the rule expression, concatenate the rule expression with the evaluated entity rule expression with "$\wedge$" and check if there is a contradiction between them. We do not store the concatenated expression. We iterate to the next conditional rule and repeat the process.

5 Evaluation

We implement RFCKG on RFC7252 on the CoAP protocol. In this section, we report and evaluate the results.

5.1 Knowledge Graph Construction

RFCKG extracts 217 rule statements in total on RFC7252, with 136 strong statements and 81 weak statements. From the rule statements, we extract and construct 28 Entity nodes with the predefined entity list and use two predefined relations "type" and "field" to construct the skeleton knowledge graph. From the 217 rule statements, we construct 319 Rule nodes and use the predefined relation "rule" to connect them to the corresponding Entity nodes. Out of the 319 Rule nodes, there are 220 plain entity rules, 41 antecedent rules and 58 consequent rules. RFCKG extracts the knowledge that describes the behaviours of entities in rules and represent it in a knowledge graph data structure, which can be easily accessed. Table 2 shows the results of our knowledge graph construction.

5.2 Contradiction Detection

For the contradictions checking, we captured 21 contradictions in total, with 16 direct contradictions and five conditional contradictions. Out of the 16 direct contradictions, one of them is true positive, which is a weak contradiction, and 15 of them are false positive. Out of the five conditional contradictions, four of them are true positive, and one of them is false positive. Table 3 shows the result of contradiction checking. Table 4 shows the five detected contradictions and their original rule statements.

Table 3. Results on contradiction detection from RFC7252

	True positive	False positive	Total
Direct contradiction	1	15	16
Conditional contradiction	4	1	5
Contradiction	5	16	21

Table 4. Detected contradictions and their original rule statements

	RS1	RS2	Type
C1	Implementations SHOULD also support longer length identifiers and MAY support shorter lengths	Note that the shorter lengths provide less security against attacks, and their use is NOT RECOMMENDED	Direct
C2	The Token Length field MUST be set to 0 and bytes of data MUST NOT be present after the Message ID field	If there are any bytes, they MUST be processed as a message format error	Conditional
C3	An option that is not repeatable MUST NOT be included more than once in a message	An option MAY be included one or more times in a message	Conditional
C4	Any attempt to supply a NoSec response to a DTLS request simply does not match the request and therefore MUST be rejected	Unless it does match an unrelated NoSec request	Conditional
C5	Implementations of this specification MUST set this field to 1 (01 binary)	Messages with unknown version numbers MUST be silently ignored	Conditional

5.2.1 Direct Contradiction

Refer to the two original rule statements for C1 in Table 4. The first rule statement describes shorter length identifiers MAY be supported. The second rule sentence describes that the use of shorter length identifiers is NOT RECOMMENDED. The variable extracted from both rule statements "support shorter length". The value from the first statement is True and the value from the second statement is False, hence it yields a contradiction. This contradiction might cause confusion for implementations for this functionality, e.g., one implementation supports shorter length identifiers and the others do not. Also, they are on the same requirement level, and one cannot overwrite the other.

5.2.2 Conditional Contradiction

For the four true positive conditional contradictions, all of them are to describe the handling of the situations that are different from the entity plain rules.

Refer to C5 in Table 4. The variable extracted from both rule statements is "version number", while the first rule statement states that it should be equal to 1 and the second rule statement states that it should not be equal to 1. The antecedent of the second rule statement is contradicting the first rule statement, but the consequent describes what should be done if it happens. Although these conditional contradictions are not real inconsistencies but to describe the error handling, they still deserve to be highlighted for developers to make sure the error handling practice is correctly followed.

5.2.3 False Positive Analysis

We now take a closer look at the false positive cases we found and understand why they happen. Recall that when we extract the variables from the rule statements, we defined 59 syntactical patterns for sentences with different syntactical structures and apply them to all rule statements. Then we normalise variables that have a similar meaning as one variable. The reason why these false positive contradictions are captured is that, some of the syntactical patterns are extracting irrelevant variables from some rule statements, and they are normalised as one variable, which is the same variable from other rule statements with different values. Hence a contradiction is observed. We examine all the false positive contradictions with their original rule statements, the extracted variables and the normalised variables. We find that all of them are captured due to this reason.

6 Discussion and Future Directions

We now discuss some limitations of this work and some possible future directions.

Entity and Relation Extraction. The construction of a knowledge graph is based on predefined lists of entities and relations, and it requires prior knowledge to determine the relation between the entity pairs. There exist tools for general purpose entity extraction and relation extraction. More advanced NLP tools, such as BERT [6] and GPT-3 [4], are language models pre-trained on large scale corpora and they perform well on a range of general tasks. A possible future direction is to take these existing models and further pre-train them with domain specific corpora, so that they would adapt their parameters to better relate different words within the domain specific language environment. Then we take these further pre-trained models and use them on the desired downstream tasks to construct the knowledge graph in a more automated way.

Co-reference Resolution and Sentence Simplification. Recall that when we construct the Rule nodes in Sect. 4.2.2, we introduce the co-reference resolution and spilt and rephrase techniques. Although they improve the process to construct the knowledge graph, they also input noise and errors into the graph

when they cannot identify the correct co-reference or split the sentence in the wrong way. The co-reference resolution tool we use is also for general purpose. The similar idea to further pre-train an existing language model to inject domain knowledge also applies here.

Introduced Noise. In our approach, although there is some noise being introduced, we argue that it is still reasonable to do so. From our true positive direct contradiction case, the original variable being extracted are "support shorter lengths" and "use shorter lengths", and they have the value True and False respectively. If we do not normalise these two variables as one, we will not find this contradiction exists. Furthermore, our work does not target to only capture precise contradictions, but to send out warnings when we suspect that there might be a contradiction. However, a possible solution to improve this might be to split the sentences into several clusters, where each cluster contains sentences that have similar syntactical structures. We might be able to describe the syntactical structure of a sentence with features like the number of syntactical roles (verbs, subjects, etc.), positions of these roles, and so on.

Implicit Condition. Recall that when we construct the antecedent Rule nodes and the consequent Rule nodes, we identify the conditions based on the explicit use of the keyword "if". There exist some situations that the condition is expressed implicitly in the rule statements. A systematic way to identify these implicit conditional relations is worth being explored.

Reasoning Scalability. We observe that the `SymPy` [20] solver we use for contradictions detection is not satisfactorily efficient even on a single specification document (RFC7252), due to the large number of variables extracted and evaluated. This indicates there would be a scalability issue when we apply this approach to a broader range of documents. A recent work by Zhang *et al.* [36] inspires us that it is possible to utilise the ability of deep neural networks for more efficient reasoning.

7 Conclusion

In this work, we propose RFCKG, a knowledge graph based contradiction detection tool for RFC documents. We implement it on RFC7252 of CoAP and successfully detect five contradictions, with one direct contradiction and four conditional contradictions. We evaluate the results and propose future directions to improve our approach to be more accurate and more automated.

Appendix A Table for Predefined Entities and Relations

Table 5. Predefined entities and relations

Entities	Relations
message, empty message, version number, token length, payload, option, option number, option delta, configuration, option length, endpoint, recipient, option value, confirmable message, acknowledgement message, reset message, non-confirmable message, message id, client, get, put, delete, server, sender, response code, proxy, uri-path option, proxy-uri option, etag option, location-path option, constrained networks, datagram transport layer security, dtls, pre-shared key, psk, raw public key, x.509 certificate, certificate, application environment, post, if-match option, if-none-match option, origin server, content-format, resource discovery, intermediary, forward-proxy, reverse-proxy, coap-to-coap proxy, cross-proxy, separate response, critical option, elective option, unsafe option, safe-to-forward option	type, field, rule, condition

Appendix B Algorithms for Extracting Rule Statements and Detecting Contradictions

Algorithm 1. Extracting rule statements from RFC

Input: Preprocessed RFC document $\mathcal{R}$
Output: strong and weak rules statements sets

1: $modal_keywords \leftarrow$ [“MUST”, ... , “MAY”, “OPTIONAL”]
2: $strong_modal_keywords \leftarrow$ [“MUST”, ..., “SHALL”]
3: $strong_rules_statements \leftarrow [\ ]$
4: $weak_rules_statements \leftarrow [\ ]$
5: $rfc_sentences \leftarrow NLTK.split_sentences(\mathcal{R})$
6: **for** $i \leftarrow 0,\ length(rfc_sentences)$ **do**
7: $\quad sentence \leftarrow rfc_sentences[i]$
8: $\quad$ **for** $j \leftarrow 0,\ length(modal_keywords)$ **do**
9: $\quad\quad keyword \leftarrow modal_keywords[j]$
10: $\quad\quad$ **if** $keyword$ **in** $sentence$ **then**
11: $\quad\quad\quad$ **if** $keyword$ **in** $strong_modal_keywords$ **then**
12: $\quad\quad\quad\quad strong_rules_statements.append(sentence)$
13: $\quad\quad\quad$ **else**
14: $\quad\quad\quad\quad weak_rules_statements.append(sentence)$
15: $\quad\quad\quad\quad$ **break**
16: **return** $strong_rules_statements,\ weak_rules_statements$

Algorithm 2. Check Direct Contradictions

```
Input: entity_rules_set, variable_symbols_set, variable_values_set
Output: direct contradictions set, entity evaluation
1: entity_evaluation ← True
2: entity_rules_evaluated ← [ ]
3: direct_contradictions ← [ ]
4: for i ← 0, length(entity_rules_set) do
5:     rule ← entity_rules_set[i]
6:     rule_items ← rule.items
7:     rule_necessity ← rule.necessity
8:     rule_operator ← rule.operator
9:     rule_evaluation ← None
10:    for j ← 0, length(rule_items) do
11:        variable, operator, value ← rule_items[j]
12:        variable_symbol ← variable_symbols_set[variable]
13:        value_integer ← variable_values_set[value]
14:        expression ← SymPy.operation(variable_symbol, operator, value_integer)
15:        if rule_evaluation is None then
16:            rule_evaluation ← expression
17:        else
18:            if rule_operator is "AND" then
19:                rule_evaluation ← rule_evaluation & expression
20:            else
21:                rule_evaluation ← rule_evaluation | expression
22:    temp_entity_evaluation = entity_evaluation & rule_evaluation
23:    if temp_entity_evaluation is False then
24:        direct_contradictions.append((entity_rules_evaluated, rule))
25:    else
26:        entity_evaluation ← temp_entity_evaluation
27:        entity_rules_evaluated.append(rule)
28: return direct_contradictions, entity_evaluation
```

References

1. Andow, B., et al.: {PolicyLint}: investigating internal privacy policy contradictions on google play. In: 28th USENIX Security Symposium (USENIX security 19), pp. 585–602 (2019)
2. Bird, S., Klein, E., Loper, E.: Natural language processing with Python: analyzing text with the natural language toolkit. O'Reilly Media, Inc (2009)
3. Bradner, S.: Key words for use in RFCs to indicate requirement levels. http://datatracker.ietf.org/doc/html/rfc2119 (1997). Assessed 04 Aug 2022
4. Brown, T., et al.: Language models are few-shot learners. Adv. Neural. Inf. Process. Syst. **33**, 1877–1901 (2020)
5. Chegini, H., Naha, R.K., Mahanti, A., Thulasiraman, P.: Process automation in an IoT-fog-cloud ecosystem: a survey and taxonomy. IoT **2**(1), 92–118 (2021)
6. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
7. Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using networkx. In: Varoquaux, G., Vaught, T., Millman, J. (eds.) Proceedings of the 7th Python in Science Conference, pp. 11–15. Pasadena, CA USA (2008)

8. Harabagiu, S., Hickl, A., Lacatusu, F.: Negation, contrast and contradiction in text processing. In: AAAI, vol. 6, pp. 755–762 (2006)
9. Honnibal, M., Montani, I.: spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing (2017)
10. Huh, S., Cho, S., Kim, S.: Managing IoT devices using blockchain platform. In: 2017 19th International Conference on Advanced Communication Technology (ICACT), pp. 464–467. IEEE (2017)
11. Khan, L.U., Saad, W., Han, Z., Hossain, E., Hong, C.S.: Federated learning for internet of things: recent advances, taxonomy, and open challenges. IEEE Commun. Surv. Tutorials **PP**(99), 1 (2021)
12. Kraus, A.: californium. https://github.com/eclipse/californium (2016). Accessed 11 Aug 2022
13. Le, D.P., Meng, H., Su, L., Yeo, S.L., Thing, V.: Biff: a blockchain-based IoT forensics framework with identity privacy. In: TENCON 2018–2018 IEEE region 10 conference, pp. 2372–2377. IEEE (2018)
14. Leiba, B.: Ambiguity of uppercase vs lowercase in RFC 2119 key words. https://datatracker.ietf.org/doc/html/rfc8174 (2017). Accessed 04 Aug 2022
15. Li, H., et al.: Improving API caveats accessibility by mining API caveats knowledge graph. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 183–193. IEEE (2018)
16. Lynggaard, P., Skouby, K.E.: Complex IoT systems as enablers for smart homes in a smart city vision. Sensors **16**(11), 1840 (2016)
17. Mahadewa, K., et al.: Scrutinizing implementations of smart home integrations. IEEE Trans. Softw. Eng. **47**, 2667–2683 (2019)
18. Mahadewa, K., et al.: Identifying privacy weaknesses from multi-party trigger-action integration platforms. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 2–15 (2021)
19. Meng, M.H., et al.: Post-GDPR threat hunting on android phones: dissecting OS-level safeguards of user-unresettable identifiers. In: The Network and Distributed System Security Symposium (NDSS) (2023)
20. Meurer, A., et al.: SymPy: symbolic computing in python. Peer. J. Comput. Sci. **3**, e103 (2017)
21. Mondal, I., Hou, Y., Jochim, C.: End-to-end NLP knowledge graph construction. arXiv preprint arXiv:2106.01167 (2021)
22. Pacheco, M.L., von Hippel, M., Weintraub, B., Goldwasser, D., Nita-Rotaru, C.: Automated attack synthesis by extracting finite state machines from protocol specification documents. arXiv preprint arXiv:2202.09470 (2022)
23. Shanthamallu, U.S., Spanias, A., Tepedelenlioglu, C., Stanley, M.: A brief survey of machine learning methods and their sensor and IoT applications. In: 2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA), pp. 1–8. IEEE (2017)
24. Shelby, Z., Hartke, K., Bormann, C.: The constrained application protocol (CoAP). http://datatracker.ietf.org/doc/html/rfc7252 (2014). Accessed 04 Aug 2022
25. Singh, A.K.: We will be surrounded by 500 billion connected devices by 2030, says anter virk of subcom. https://opportunityindia.franchiseindia.com/article/we-will-be-surrounded-by-500-billion-connected-devices-by-2030-says-anter-virk-of-subcom-35012 (2022). Accessed 28 Aug 2022
26. Soares, L.B., FitzGerald, N., Ling, J., Kwiatkowski, T.: Matching the blanks: distributional similarity for relation learning. arXiv preprint arXiv:1906.03158 (2019)

27. Tian, C., Chen, C., Duan, Z., Zhao, L.: Differential testing of certificate validation in SSL/TLS implementations: an RFC-guided approach. ACM. Trans. Softw. Eng. Methodol. **28**(4), 1–37 (2019).https://doi.org/10.1145/3355048
28. Uddin, H., et al.: IoT for 5g/b5g applications in smart homes, smart cities, wearables and connected cars. In: 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), pp. 1–5. IEEE (2019)
29. Wang, K., Bai, G., Dong, N., Dong, J.S.: A framework for formal analysis of privacy on SSO protocols. In: Lin, X., Ghorbani, A., Ren, K., Zhu, S., Zhang, A. (eds.) SecureComm 2017. LNICST, vol. 238, pp. 763–777. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78813-5_41
30. Wang, Q., et al.: {MPInspector}: A systematic and automatic approach for evaluating the security of {IoT} messaging protocols. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 4205–4222 (2021)
31. Wang, Q., Mao, Z., Wang, B., Guo, L.: Knowledge graph embedding: a survey of approaches and applications. IEEE Trans. Knowl. Data Eng. **29**(12), 2724–2743 (2017)
32. Xiao, L., Wan, X., Lu, X., Zhang, Y., Wu, D.: IoT security techniques based on machine learning: how do IoT devices use AI to enhance security? IEEE Signal Process. Mag. **35**(5), 41–49 (2018)
33. Xie, D., et al.: DocTer: documentation-guided fuzzing for testing deep learning API functions. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 176–188 (2022)
34. Xie, F., et al.: Scrutinizing privacy policy compliance of virtual personal assistant apps. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE) (2022)
35. Zhang, B., Xu, Y., Li, J., Wang, S., Ren, B., Gao, S.: SMDM: tackling zero-shot relation extraction with semantic max-divergence metric learning. Appl. Intell. 1–16 (2022). https://doi.org/10.1007/s10489-022-03596-z
36. Zhang, C., et al.: Towards better generalization for neural network-based sat solvers. In: Gama, J., Li, T., Yu, Y., Chen, E., Zheng, Y., Teng, F. (eds) Advances in Knowledge Discovery and Data Mining. PAKDD 2022. LNCS, vol. 13281. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-05936-0_16